

ECE 1786 Lecture #6 Part 1Work-in-Flight: Assignment 3 - Training & Using Transformer, due Mon Oct 20

- Team forming should be done; I need to assign Mentors
 - Only 26 teams have filled out team form! Should be 32!
 - Team Forming Link: <https://forms.office.com/r/kHnczepPhK>
- Project approval-in-principle form: <https://forms.office.com/r/FNx9t3uLis>
 - Due October 21;

Last Day: Core Mechanisms of Transformer Block & Assignment 3

Today: 1. Language **Generation** using Transformers
 2. Agentic Project Ideation, Approval-in-Principle and Proposal

Language Generation Using Transformer Language Models

Recall: A language model is trained to predict the **next token** that comes after a given input sequence of tokens. (From now on, using token, not word)

So, if you can do that, then you can also produce a sequence of output tokens, one at a time, by taking each predicted token, **appending it** to the input sequence of tokens & then predicting/generating the next token after that, and so on.

⇒ This is called **Auto-regressive** generation, a term that comes from the world of sequence predictions. (Not an obvious name..) See Jurafsky section 10.1

Here is the what I'd call the "Auto-regressive loop:" (maybe obvious, but is very important):

e.g. If the input started as: "The clean river flowed"
 Call model to infer & generate next word: e.g. "into"

Then, the next input to the model is "The clean river flowed into"

Next word might be: "the"

Next input would be "The clean river flowed into the"

Generate next word, and so on. This is how chatGPT delivers what you ask for.

Each generation of a word is pretty expensive - because a full inference is run just to get one token, of a very large model- why LLMs are said to be hurting the planet, as quite a bit of energy is consumed in each inference - need lots!!

Example Generation from GPT-2

See: <https://banana-projects-transformer-autocomplete.hf.space/doc/gpt2-large> (GPT2 Large only)

1. From GPT-2 medium (mid-sized GPT-2 in terms of # parameters).

Input: The clean river flowed

Generation Parameters (Unknown: i.e. Temp, Top-P, etc., max tokens generated)

Output: The clean river flowed in steady but strong currents.

After a week of hard-boiled fish hunt, all was tranquil again. They went out in a group on the small river and ate their dinner along the banks with some of the fish

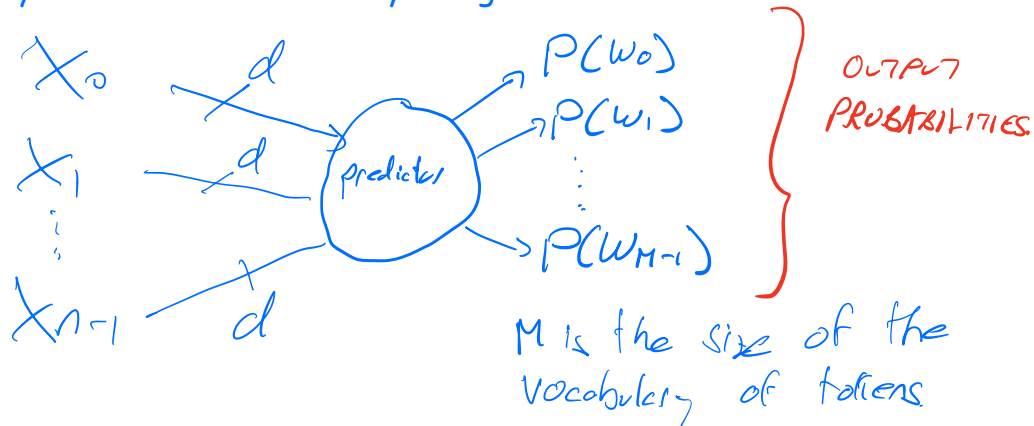
2. From GPT-2 XL (largest GPT-2).

Input: The clean river flowed

Generation Parameters (Unknown)

Output: The clean river flowed. We walked on to the other side with the people we left behind. We found a small restaurant with a bench near the river — a small oasis at the end of the world, really.

Recall the specific input & output of the language model/transformer: given an input sequence of token embeddings, X_0, X_1, \dots, X_{n-1} , output the probability of every token in the vocabulary being the next token:



So, for a given input sequence, which **single** token is selected as the output?

The process of selecting the token, based on the output probabilities, is called **decoding**. This word is an unfortunate choice, as it is confused with decoders that are quite something else in this field. (So make sure you pronounce the "ing" on decoding). We speak of the 'decoding' algorithm, i.e. given $P(W_i)$, select one of the W_i as the output.

What do you think the best approach/decoding algorithm is?

This needs some careful thought. We don't just want the best **next** token, but the best **sequence** of output tokens. That does depend on the next token, but also what will come after the token.

What makes the sequence best? i.e. what are its properties?

The full input/output sequence should be:

- Grammatical
 - Make **sense given the input context**
- But now we also want it to be correct, smart, helpful

6-4

Method 1: **Greedy** Decoding: select the highest probability word.

- You will see this in Assignment 3 Section 2
- Greedy **does not** work well in general - it picks obvious words, but these often lead to boring, uninteresting sequences of words; also repetitive
- Greedy may choose the most likely next token, but does not result in the most likely sequence of generated tokens
- Gets stuck in a highly local optimum in the space of all possible generated sequences; **at least this was true; might be different with new models**

We can express this issue mathematically as follows: Given an input sequence of token embeddings $X_0 \dots X_{n-1}$ we want the generated sequence of output tokens $Y_0 \dots Y_{g-1}$ of g tokens to be the most likely sequence

i.e. we want $P(Y_0) \times P(Y_1) \times \dots \times P(Y_{g-1})$ to be maximized.

But, we don't know $P(Y_1)$ when selecting Y_0 (or Y_j , $j > i$ when selecting Y_i)

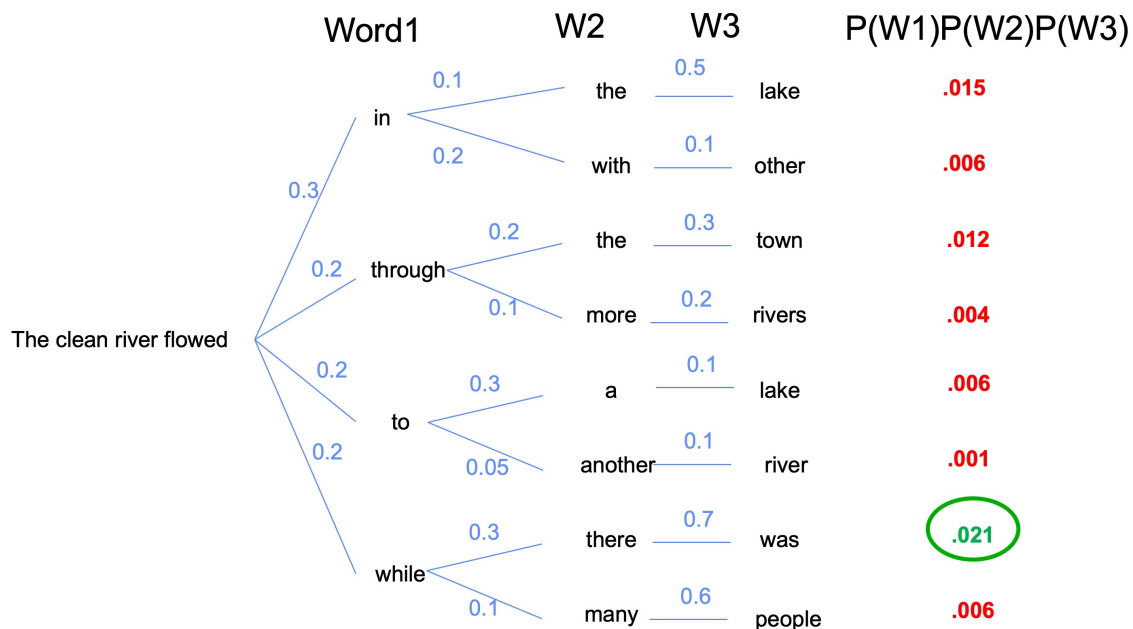
This is a hard problem because there are $M^{**}g$ possible sequences of g output tokens, which is a very big exponential. Here M is the size of the vocabulary

- If $M = 50000$, $g=20$, $50000^{**}20$ is huge.
- Worse, each **token output** is a full forward inference of the model!

As shown in Jurafsky Section 10.2, think of the auto-regressive selection of the output sequence as a tree with probabilities at each layer.

Input: the clean river flowed

Outputs, produced from many invocations of inference of the trained model:



Notice how the most probable Word 1 doesn't lead to the most probable sequence of Word 1, Word 2, Word 3. (.015 or .006 vs. .021)

- o But determining the highest probability sequence is very hard.

Method 2: **Beam Search** is a heuristic that prunes the full search tree a lot.

General Description:

- o Walk down the tree, keeping the K-most probable sequences.
- o At each level of the tree, consider top V possible next words for each of the K sequences. (Means you need K separate inferences instead of min 1)
- o Compute the full sequence probability of those possible K x V sequences
 - Keep the K highest
- o repeat until have generated the number of desired tokens (or hit stop token)

Method 3: Sampling (most commonly used)

Given: The set of output probabilities $P(W_0), P(W_1), \dots, P(W_{M-1})$

Select: The next token through a "random process" in which the probability of selecting token W_i is $P(W_i)$ — how?

- In words: toss a many-sided weighted die, where the weights of each side (word) are the probability of the word.
- So, although the highest probability word is the most likely to be chosen, it isn't necessarily chosen - depends on the relative probabilities.

Illustration with an example:

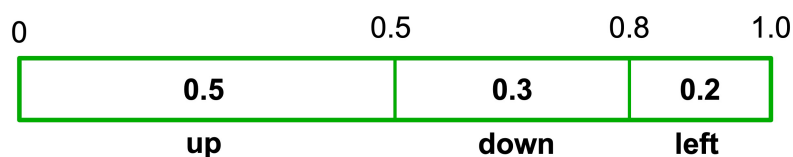
Consider a 3-token vocabulary: up, down, left

Assume the neural network has generated these probabilities for next word:

- $P(\text{up}) = 0.5$
- $P(\text{down}) = 0.3$
- $P(\text{left}) = 0.2$

We want random process that selects **up** with probability 0.5, **down** with prob 0.3 and **left** 0.2

We can do that by 1) visualizing the probabilities as shown below, and 2) generating a (uniformly distributed) random number, R , between 0 and 1, inclusive:



So:

- If $R \leq 0.5$ select **up**
- If $0.5 < R \leq 0.8$ choose **down**
- If $0.8 < R \leq 1.0$ choose **left**

} up will be chosen with probability 0.5, left with prob 0.2

This random process has a nice side-effect: if you don't like the output sequence that you get, you can just try again and get a new one.

This reflects the fact that there are many ways to answer a given question, or to create language.

However, this randomness is also part of the source of the 'hallucinations' that you've probably heard of from chatGPT/LLMs. Some bad luck on the first word could just send the answer in the wrong direction!!!!

This method is the most widely used; you used to see parts of it on the GPT-3/4 playground, and in the code in Assignment 3, Section 2.

There are several variations to know about (& you'll see in the MinGPT code):

- Rather than select from all M tokens in the vocabulary, only select from the top K most probable tokens. (Called **top-k sampling**)
- More commonly used: **top-p sampling**: only select from the top words that all together have the sum of probabilities = p (or closest). $0 \leq p \leq 1$
 - If set $p = 1$ that means use all M tokens in the vocabulary
 - Often $p = 0.8$
- There is one more important adjustment to this process that is important:
 - The output token probabilities from the model are **adjusted** to control whether the generated sequences are more or less creative/diverse.
- It is done with a parameter, t , called the **Temperature**
- A high T gives more diverse words, done by adjusting probs before sampling
- $T = 1$ is 'normal' - the probabilities are unchanged
- $T > 1$ makes less probable words more likely
- $T < 1$ makes more probable words more likely
- $T = 0$ makes the decoding greedy

The probabilities are adjusted during the Softmax output computation of the probabilities, as shown in this equation:

$$p(w_i) = \frac{\exp\left(\frac{l_i}{t}\right)}{\sum_{all\ i} \exp\left(\frac{l_i}{t}\right)}$$

Where the l_i are the final logits produced by the network. (not the attention softmax)

Other notes:

- Often a combination of top-p & temperature are the commonly used generation parameters
- There is also a repetition penalty - e.g. divide l_i by 1.3 if the token corresponding to l_i has already been used in this generation

DEMO of GPT-3/4 on playground (vs. ChatGPT) & parameters

- Turn on "Show probabilities" with the completion interface, effect of T, max tokens; show 'code' to generate using API; see <https://platform.openai.com/playground/complete>

Here is a screenshot of the legacy completion models from OpenAI that let one show the probabilities of the words sampled from at each word generated

- just one is being showed below, the alternative higher-probability words to the word 'not' after the words "It is ..."

The clean river flowed 80 miles.

It is not possible to determine the speed at which the clean river flowed without knowing the specific location and conditions of the river. Different sections of a river, such as depth, slope, and obstructions. Additionally, rivers can flow at different speeds at different times due to

sea: not = 59.99%

- impossible = 11.35%
- unclear = 9.00%
- difficult = 7.62%
- possible = 3.72%
- important = 2.99%
- a = 0.60%
- gramm = 0.57%
- likely = 0.54%
- unknown = 0.43%

Total: -0.51 logprob on 1 tokens
(96.78% probability covered in top 10 logits)

Warning: Your text ends in a trailing space, which causes worse performance due to how the API splits text into tokens.

Submit

Model

gpt-3.5-turbo-instruct

Temperature 1

Maximum length 2048

Stop sequences

Enter sequence and press Tab

Top P 1

Frequency penalty 0

Presence penalty 0

Best of 1

Inject start text

Inject restart text

Show probabilities

Full spectrum

Part II: Project Ideation and Scoping